A Trace-Driven Comparison of Algorithms for Multi-Process Prefetching and Caching

Andrew Tomkins R. Hugo Patterson Garth Gibson

September, 1996
CMU-CS-96-174

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was supported in part by the Data Storage Systems Center under National Science Foundation grant number ECD-8907068, in part by Advanced Research Projects Agency contract DABT63-93-C-0054, and in part by generous contributions from the member companies of the Parallel Data Consortium. At the time of this writing, these companies include Hewlett-Packard Laboratories, Symbios Logic Inc., Data General, IBM Corporation, Seagate Technology, EMC Corporation, Storage Technology Corporation, and Digital Equipment Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

Keywords: File Systems, Parallel I/O, Prefetching, Informed Caching, Hints, TIP, LRU-SP

Abstract

Recently two groups of researchers have proposed systems that exploit application knowledge to improve I/O performance. Both systems use application knowledge to prefetch data thereby masking I/O latency and to improve file buffer cache performance thereby avoiding slow I/O accesses altogether. Unfortunately, published studies of these two systems are incomparable. This technical report is a follow-on to a paper to appear in OSDI96 comparing the TIP2 system of Patterson, Gibson, et al, and the LRU-SP system of Cao, Felten, Karlin and Li, co-written by the two groups. The OSDI paper considers the case of a single process with full advance knowledge of requests. In this technical report we consider multiple processes, each of which has either full advance knowledge (complete hints) or no advance knowledge (no hints).

Our results can be summarized as follows: the cost-benefit analysis of TIP2 allows better performance when optimal buffer allocation does not correspond to process consumption rates.

1 Introduction

Traditional filesystems wait until an application requires data, and then initiate an I/O request. If this request is not present in the buffer cache, it generates a disk access and ejects a buffer under the LRU replacement policy. Recent work on integrated prefetching and caching suggests that both phases of this procedure can be improved dramatically using application-disclosed access information. First, I/O stall time can be reduced by prefetching data. Second, buffer cache replacement decisions can combine traditional LRU information and application-provided information about future requests from hinting processes. Two recent systems have presented approaches to both aspects of this problem.

The TIP2 system of Gibson, Patterson et al [PGG+95] ("Transparent Informed Prefetching", second generation) addresses this problem via an integrated cost-benefit analysis of the value of maintaining certain data in the buffer cache. Prefetching is initiated when ejecting a cached block has lower cost than the benefit accrued by loading a block that is not present in the cache. Whenever a block must be ejected, the global lowest-cost block is chosen.

The LRU-SP system with controlled-aggressive prefetching (hereafter referred to simply as LRU-SP), presented by Cao, Karlin, Felten and Li [CFKL95b, CFKL95a, CFL94a, CFL94b, Cao96], addresses the problem via a two-level procedure. Buffers are allocated among processes by the kernel using the LRU-SP procedure, which ejects a block from the process containing the globally least-recently-used block. This process may, however, suggest a more appropriate block to eject based on information provided by the application. If the application has detailed knowledge about its access patterns, the process' buffer manager will use the *controlled-aggressive* algorithm to make process-local decisions about prefetching and replacement.

This technical report is a follow-on to a paper that will appear in OSDI96 [KTP+96]. The original paper is co-written by the authors of the two systems studied in this report, and considers the case of a single process with complete knowledge of future accesses. Prior to this paper, published studies of the two systems had not been comparable. Differences in hardware, both in the processor and the I/O subsystem, as well as in the benchmarks used to evaluate the systems have made it difficult to understand the differences between the algorithms. This report contains a follow-on study of the multiple process case, assuming that each process gives either no hints or complete hints about its accesses.

An algorithm for prefetching and caching must (either implicitly or explicitly) solve two distinct problems. First, the algorithm must determine when a process should use a particular buffer for prefetching and when it should use the buffer to cache data for reuse. Second, the algorithm must decide how buffers should be allocated among various processes running simultaneously. We consider only the second problem, as [KTP+96] considers the first problem in some detail.

We compare these two approaches using trace-driven simulation. We have developed a multi-threaded disk-accurate simulator built upon RaidSim, a platform for modeling various flavors of RAID disk arrays. Our simulator is driven by sets of per-process traces of disk operations and (process) compute time between operations; it performs context switches according to an internal scheduler based on the Berkeley Sprite operating system, swapping out traces representing processes that are, for instance, waiting for I/O.

Section 2 describes the two algorithms in more detail, and describes our tracing and simulation environment. In Section 3 we give synthetic traces highlighting some advantages of cost-benefit analysis. Then in Section 4 we consider application traces of two hinting processes, and in Section 5 we consider traces of a hinting process running alongside a non-hinting process. Section 6 contains some sensitivity analysis of the earlier results. And finally, Section 7 presents our conclusions.

Throughout, the focus is on the performance of the algorithms for prefetching and caching. To maintain this focus, we sidestep various issues. First, we use SCAN queueing since it is a common disk scheduling algorithm, and we do not give a detailed analysis of the effects of disk queueing. Furthermore, we restrict our attention to applications that provide lists of their future accesses. Both systems support this model of operation, and all applications we have considered are capable of providing hints in this form. Thus, we avoid a discussion of the merits of the other less powerful forms of cache management advice supported by LRU-SP. Finally, we do not address issues of missing or incorrect hints; the problem is significant, but is beyond our scope.

2 Background

In this section we give a brief description of the two systems, and then provide information about our simulator and our trace-collection mechanism.

2.1 TIP2

The TIP2 system does not explicitly distinguish between local, single-process buffer allocation and global allocation among processes. Instead, it uses cost-benefit analysis applied to all buffers when making allocation decisions. Nevertheless, it is reasonable and instructive to consider these two case separately.

When a system is only running a single hinting process, TIP2 balances the benefit of using a buffer for prefetching against the cost of ejecting a block that hints indicate will be reused. In estimating the benefit of prefetching, TIP2 assumes sufficient I/O bandwidth to avoid disk congestion and that all disk accesses complete in time T_{disk}. Since reading a block from the buffer cache consumes a non-zero amount of time, T_{hit}, there is no benefit from initiating prefetches more than $T_{\text{disk}}/T_{\text{hit}}$ accesses in advance. This distance is called the *prefetch horizon*. On the flip side, the only cost of ejecting a block is the CPU overhead of prefetching the block back later since with sufficient bandwidth, this can be done without stalling for the read. TIP2 averages this CPU cost over the number of accesses till reuse.

The net result of this caching and prefetching algorithm is that, for a single process, TIP2 prefetches in order up to or close to the *prefetch horizon*, never ejects one block to prefetch another block that will be used after the first one, and always chooses for ejection the block whose next use is farthest in the future. When the assumption of adequate I/O bandwidth is not met, for example when there is only a single disk, the limit on prefetching can cause the disk to go idle during lulls in I/O activity when the disk could be used to prefetch for an upcoming burst of activity.

When there are multiple hinting processes, TIP2 scales the benefit and cost estimates described above by the access rates of the processes. Thus, TIP2 prefetches further for a process that is consuming data at a high rate. Similarly, TIP2 is willing to cache more blocks for a higher data-rate process.

To estimate the cost of ejecting a non-hinted block, TIP2 keeps non-hinted blocks in a separate LRU queue and monitors data reuse to estimate the cache hit rate as a function of LRU queue size. The cost of ejecting a non-hinted block is the estimated increase in cache misses and associated I/O stalls that would result from a smaller LRU queue. Thus, TIP2 tends to grow the LRU queue at the expense of hinters when doing so would increase the non-hinting cache hit rate and it tends to shrink the LRU queue freeing buffers for hinted caching when doing so would not hurt the performance of non-hinters.

2.2 LRU-SP with Controlled- Aggressive Prefetching

In the LRU-SP system, individual hinting processes use the controlled-aggressive algorithm to control their personal prefetching and caching behavior. The kernel implements the LRU-SP algorithm to allocate buffers among processes. We describe these two components separately, and then discuss their interaction.

Controlled-aggressive ejects block e to prefetch block p if p is not in memory, e is in memory, p occurs before e in the hint stream, and the disk is idle [CFKL95b]. Essentially, the algorithm prefetches as aggressively as reasonable, subject to disk bandwidth availability. This algorithm is very similar to TIP2's, but differs in two respects. First, it does not stop prefetching at a prefetch horizon, but instead may prefetch infinitely far in advance when disk bandwidth is available.

As we will see, this gives controlled-aggressive greater resilience in the face of bursty workloads, but does so at the risk of thrashing the cache. The second difference is that controlled-aggressive waits till the disk goes idle to queue further prefetches. In practice, the creators of this system recommend batching requests to the disk. In our implementation of controlled-aggressive, we issue new prefetches when there are less than 16 prefetch requests queued. This is enough to ensure high utilization of the four disks in the simulated disk array.

The authors of [CFKL95b] show that on a single disk, if the I/O time is F times greater than some fixed interaccess CPU time, and the buffer cache contains K buffers, then controlled-aggressive is guaranteed to be within a multiplicative factor of $1 + F/K$ of optimal. For instance, with a 1500-buffer cache and a disk that can always serve a request within 50 read hits, the total time is guaranteed to be within about 3% of optimal for any access sequence.

The controlled-aggressive algorithm was originally designed for use with a single disk. Later work applied the algorithm to multiple disks and proposed the reverse-aggressive algorithm to ensure efficient utilization of the multiple disks [KK96]. Their results, and the results of [KTP+96], indicate that controlled-aggressive performs almost as well as reverse-aggressive when data is striped over the multiple disks as it is in this study.[1] Therefore, we do not implement the reverse-aggressive algorithm.

The LRU-SP buffer cache management algorithm allocates buffers among competing processes. It accepts advice about buffer replacement from individual processes, and induces a partition of the buffer cache among the processes. The goal of the algorithm is to allocate buffers with the same fairness as the global LRU queue does.

When a buffer is required, either for a demand read or for a prefetch, LRU-SP finds the process that owns the global least-recently-used block of the buffer cache. That process is asked to give up a block. The process may simply choose to give up the LRU block itself, or may make a different decision based on information from the application. If all processes agree to give up the block suggested by the kernel then LRU-SP becomes the standard LRU buffer replacement policy. However, if a process chooses to give up an alternate block, that process will again hold the global LRU block and would be asked to give up another block when the kernel requires one. A scheme called *swapping* removes this difficulty, but introduces the possibility that a malicious process could give up blocks so as to retain an unfair share of the total buffer cache. Another scheme called *placeholders* guards against such behavior.

As originally conceived, controlled-aggressive operated on the entire buffer cache. When combined with LRU-SP, controlled-aggressive operates as if its partition

[1]When the amount of bandwidth available is extremely high controlled-aggressive performs little caching and incurs additional driver overhead; this is the only common situation we have identified in which reverse aggressive performs better.

is the entire cache. It initiates a prefetch only if there is a block in its partition that it would be willing to eject. However, when it prefetches, it asks LRU-SP to allocate a buffer. If the buffer at the head of the global LRU list belongs to a different partition, then the prefetch will have the effect of growing its partition.

2.3 Simulation Environment

Our simulator is built on top of the Berkeley RaidSim [CP90, LK91] simulator, as modified at CMU. RaidSim can simulate various flavors of RAID disk arrays using a disk geometry module to determine disk access times. In our simulations, we run with data striped over an array of disks (from 1–10 disks), with no parity and a stripe unit of 1 block. The geometry module simulates the performance of the HP97560 disk drive. RaidSim also supports various forms of prioritized disk queueing built around the CVSCAN scheduling discipline. In this paper, we will occasionally mention effects of disk queueing, but unless otherwise stated all results use SCAN disk queues.

We augmented RaidSim to include a buffer cache module layered on top of the disk array and implemented modules for both TIP2 and LRU-SP. We also added a module to drive RaidSim from scripts instead of relying on randomly generated workloads. We took advantage of RaidSim's support for multiple threads to allow the concurrent simulation multiple separately-scripted processes.[2] We drove the simulator with traces taken at CMU, and with existing traces used by the authors of [CFL94b].[3] We now describe these two sets of traces in more detail.

2.3.1 CMU Traces

To drive the simulator, we traced applications using the DFSTrace [MS96] tracing tool running in Mach 2.6 [M. 96] on a DECstation 5000/200. We augmented DFSTrace to collect buffer cache operations and scheduler activity, resulting in a set of detailed trace files on the order of 1M per ten seconds of computation.

We post-processed these traces into scripts that capture the behavior of an individual process or a process group. Each script record contains the process CPU time, derived from the scheduler trace records, and a primitive operation such as "read a block", "read an inode", "read then asynchronously write a block" , or "set the dirty bit for a block". Because these scripts capture the behavior of a process and not the whole system as a black box, we can run multiple scripts simultaneously on the simulator and explore the interactions of multiple processes.

[2] This simulator is available to the research community; send mail to the contact author.

[3] Thanks to Pei Cao for making these traces available, and to Tracy Kimbrel for providing them.

There are two effects that this simulation environment fails to capture. The first is memory contention. The traces do not include memory usage information, nor do the scripts include paging activity. When multiple processes are run together, they may compete for virtual memory causing an increase in paging activity that would not occur in our simulator. The second effect is the process CPU time required to initiate a disk access. The simulator adds this time when it initiates an access, but any CPU time that the traced process spent initiating accesses is not subtracted from the script records. This has the effect of slightly dilating the process CPU time for accesses that caused disk accesses on the original system.

We adopted only a single trace from this suite:

XDS : a 3-D data visualization program, we traced XDataSlice generating 25 planar slices through a 3-D dataset represented by a 64M file. This trace is low-reuse, and also exhibits poor striping on even numbers of disks.

2.3.2 Wisconsin Traces

We also consider a set of read-only traces with process CPU time collected on a DECstation 5000/200. The running time of these applications is dominated by disk read accesses. We used the following three traces from this suite:

CSCOPE1,CSCOPE2 : an interactive C-source examination tool written by Joe Steffen. CSCOPE1 is a trace of a search for eight symbols in an 18M software package. CSCOPE2 is a search for four text strings in the same 18M software package. Cscope reads multiple files sequentially, and will read them multiple times when there are several queries. Both traces have a high degree of re-use.

GLIMPSE : a text information retrieval system from the University of Arizona, search for four keywords in a 40M snapshot of news articles. It builds approximate indexes for words to allow both relatively fast search and small index files. The result is that the index files are accessed repeatedly, whereas the data files are accessed infrequently. We characterize this trace as medium re-use.

2.3.3 Single Process Performance

For comparison purposes, Figure 1 gives the results of both algorithms running each of the traces separately. As a convention, if we are comparing TIP to LRU-SP, we will always graphs TIP on the left and LRU-SP on the right. Table 1 gives some statistics about the traces, including average compute time (ignoring

Trace	# reads	# distinct reads	compute/read	time/read
Cscope1	8673	1073	2.87	2.93
Cscope2	20197	2462	1.83	3.78
Glimpse	27963	5247	0.91	4.12
Xds	32371	5853	0.92	2.45

Table 1: Statistics For Traces

driver time) per request, and average total time per request, computed as an average of TIP2 and Aggressive's performance.

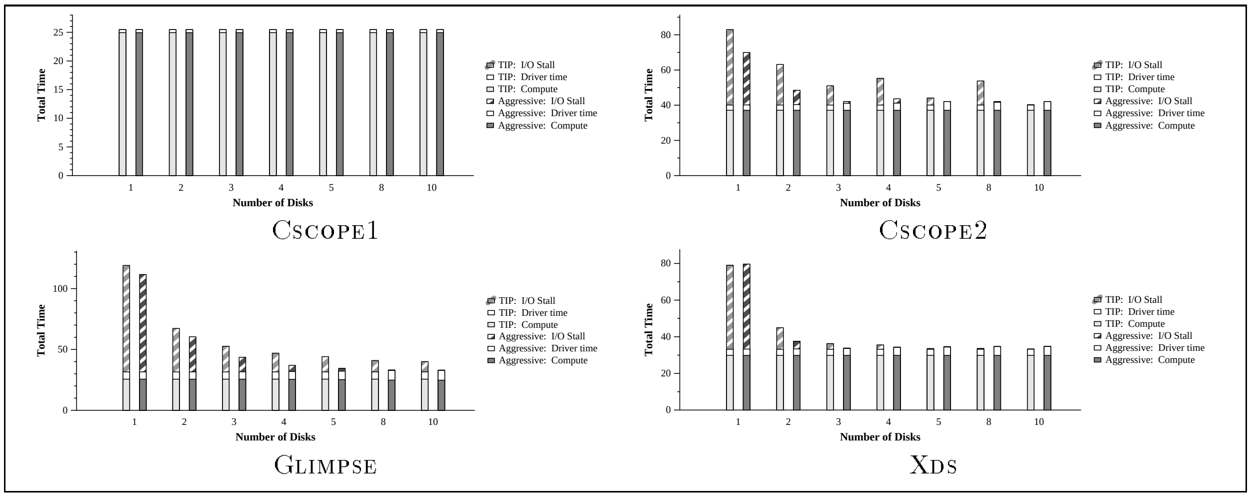


Figure 1: Single-Process Trace Results

2.3.4 TIP2 Parameter Settings

Finally, the TIP2 cost-benefit calculations require a set of system-specific parameters. In our system, these parameters are as follows: T_{hit}, the time to read an 8 KB block from the buffer cache, is 425 μs; T_{driver}, the time to initiate a disk access, is 500 μs; and T_{disk}, the disk access time, is 29 ms for non-sequential I/O. These parameters yield a *prefetch horizon* of 68 for the system.

3 Synthetic Workloads and Caching

In this section we give combinations of real and synthetic traces to show the various forms of cache contention that can arise with multiple processes. We consider a hinting process running alongside a non-hinting process.[4] For our first two cases we consider situations in which the hinting process has greater use for cache buffers than the non-hinting process. In both cases we consider a hinting process that loops sequentially through a 500-block dataset many times. We consider both a high-reuse and a low-reuse non-hinting process running alongside.

In the next two cases the non-hinting process has more use for the buffers than the hinting process. Clearly, the non-hinting process in both these cases must have high re-use. We consider both a low-reuse hinter, and a high re-use hinter. The former case, in which a high-reuse non-hinting process is running with a low-reuse hinting process, demonstrates another pitfall of the multiple-process case: how should the cache manager handle blocks that have been prefetched, read, and have no further hints.

3.1 Hinting High Re-use Process Taking Buffers From a Non-hinting Low Re-use Process

Consider a program that cycles through 500 blocks repeatedly, computing for 5ms between accesses. We assume that this program gives complete hints. Simultaneously, a low re-use process is running that does not give hints. This process issues a request for a new block every 5ms. Figure 2 gives the results for various cache sizes on a single disk.

The first thing we notice in this situation is that, even though the initial cache size is not large enough to hold the working set of the repeating application and the incoming blocks of the low re-use application, there is still a great deal of cache use by both algorithms. This is due to informed caching — we will see later that, as expected, a non-hinting application that cycles, with a working set too large to fit in cache, derives no benefit at all from caching.

As noted above, LRU-SP tends to allocate the cache based on the relative access rates of the two processes, while TIP splits the cache based on an estimate of the value of its buffers. Thus, LRU-SP chooses to cache only a portion of the working set of the hinting application, and devotes resources to caching blocks from the low re-use application.

3.2 Hinting High Re-use Process Taking Buffers From a Non-hinting High Re-use Process

We consider the same hinting program: a cyclic application issuing requests to a 500 block working set with 5ms of computation between each request. But

[4] We could have presented similar situations with two hinting processes, but the situation we study highlights the difficulties that arise when one process does not have perfect knowledge.

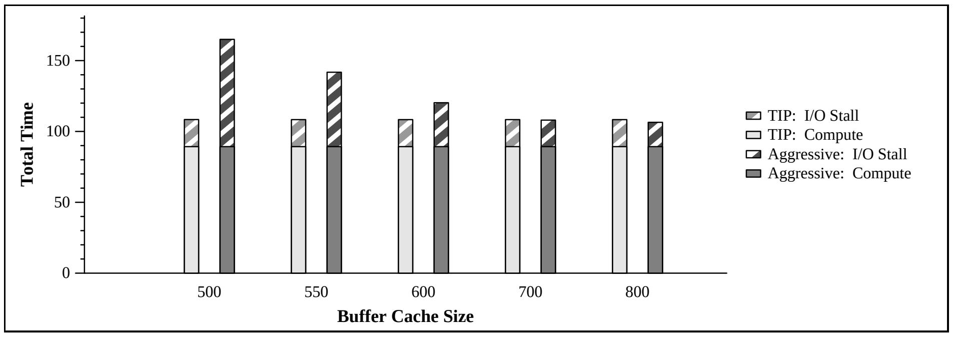


Figure 2: Hinting High Re-use Process Taking Buffers From a Non-hinting Low Re-use Process

instead of a low re-use process running in the background without hints, we consider another high re-use process. In particular, we consider another looping process with a larger working set (1200 blocks) that issues requests every 5 ms. The results are shown in Figure 3.

The analysis here is very similar to the analysis of the previous section. Since both TIP2 and LRU-SP do not have enough buffers to cache the entire working set of the non-hinting application, no caching benefit is attained.

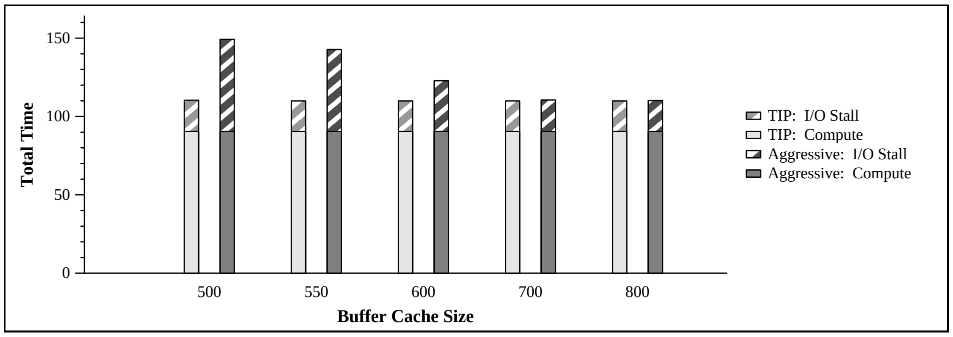


Figure 3: Hinting High Re-use Process Taking Buffers From a Non-hinting High Re-use Process

3.3 Non-hinting High Re-use Process Taking Buffers From a Hinting Low Re-use Process

We now give two examples from the opposite situation, in which the correct decision is to use the cache to hold data from the non-hinting process. In both cases we consider a slow process cycling repeatedly through 500 blocks at the rate of one block every 10 ms, without giving hints. First we consider a low re-use hinting process. In particular, we consider the XDS process described earlier. The results are shown in Figure 4.

As expected, we note that with 500 cache buffers, neither algorithm derives any caching benefit because some blocks must be used to hold data for the hinting process. Once we have 550 cache buffers, however, TIP's estimators conclude that caching data for the non-hinting process is more important than prefetching ahead for the hinting process. LRU-SP splits the buffer cache according to the relative rates of the processes, giving much of the cache to the hinting process. The relatively high rate of XDS means that the cache must become quite large (1500 buffers) before LRU-SP will allocate sufficient buffers to hold the working set of the non-hinting application.

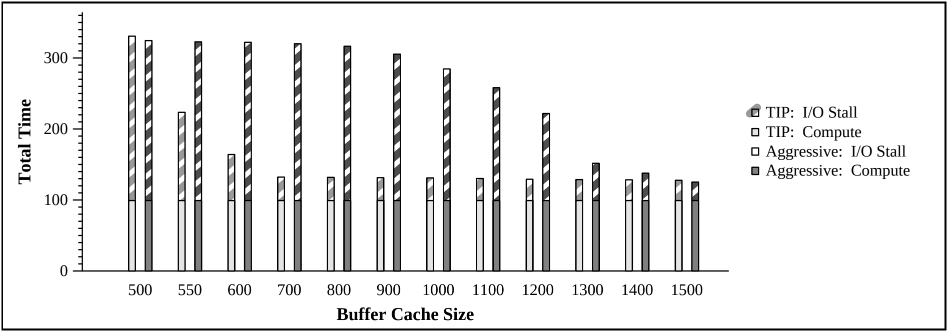


Figure 4: Non-hinting High Re-use Process Taking Buffers From a Hinting Low Re-use Process

This particular case raises an additional issue for TIP. Once the blocks for the low re-use process have been read, there are no future hints for them, so their status becomes uncertain. If we allow them to be evicted immediately in order to improve caching of the non-hinting process, it is possible that an unhinted read for those blocks will arrive, creating unnecessary stall. On the other hand, if we place the blocks into the LRU queue after they have been read, we will be more robust to unhinted requests but we may not make good use of the cache. Figure 5 shows the results if we adopt this more pessimistic policy. Since TIP is placing prefetched buffers directly into the LRU queue, XDS ends up owning a substantial fraction of the LRU queue, and performance becomes similar to LRU-SP.

In the future, we hope to evaluate the possibility of incorporating a second queue for these blocks, akin to the LRU queue but for prefetched blocks with no remaining hints. If it turns out that these blocks are never requested then this queue would be willing to give up all of its blocks to cache other data. On the other hand, if an application hints a read to a block, then reads it several more times without hints in the near future, this second queue would discover the correct behavior of holding hinted blocks for a short period, then allowing them to be discarded.

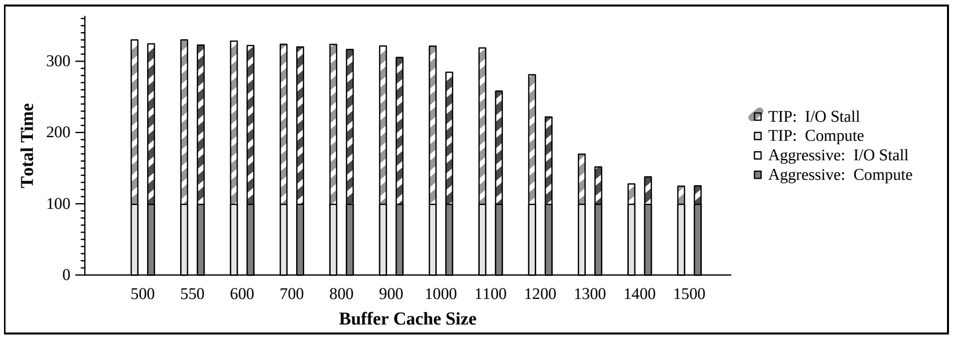


Figure 5: Non-hinting High Re-use Process Taking Buffers From a Hinting Low Re-use Process, Buffers Enter LRU Queue

3.4 Non-hinting High Re-use Process Taking Buffers From a Hinting High Re-use Process

Finally, we consider the remaining case. The same slow non-hinting process runs, cycling through 500 blocks with one request every 10 ms. Alongside, we run another cyclic process. In order to demonstrate TIP's robustness to varying data rates, we allow the hinting process to run faster, issuing requests every 2ms, and cycling through 1200 blocks. The results are shown in Figure 6.

LRU-SP performs optimal cache replacement of the blocks it caches for the faster process. Still, in this case, it turns out to be more effective to hold the entire working set of the smaller non-hinting process in the cache, freeing disk bandwidth for the hinting process.

3.5 Different Reasons to Cache

The figures above demonstrate several different forms of beneficial caching an algorithm must be able to uncover. In the first two examples the LRU cache was not providing hits, so the buffers should instead the dedicated to the hinted cache where sufficient re-use exists to make a difference. In this way, the bandwidth requirements of the hinting process can be satisfied from the cache, and the disk bandwidth can be dedicated to the non-hinting process.

In the third example the LRU cache is capable of holding the entire working set of the non-hinting process. On the other hand, the same buffers could also be

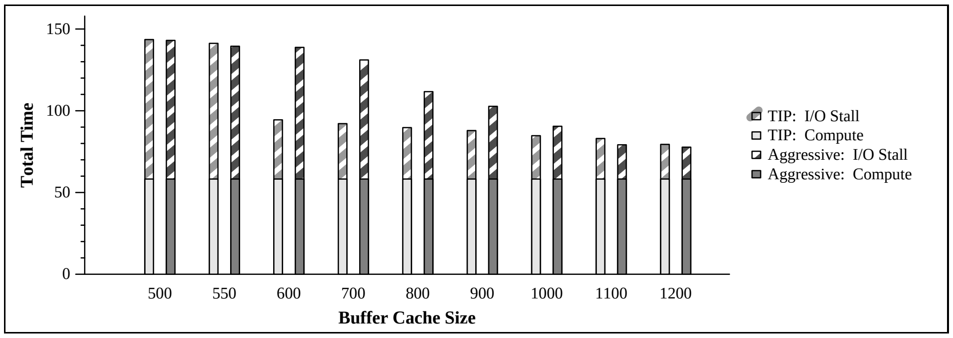


Figure 6: Non-hinting High Re-use Process Taking Buffers From a Hinting High Re-use Process

used to provide some caching for the hinting process. However, by dedicating the disk to the prefetching process and caching the non-hinting process, the needs of both applications can be met.

Finally, the last example is a case in which there is no advantage to caching data for the hinting process, and the buffers should be dedicated instead to hold the working set of the non-hinting process.

4 Application Traces: Hinting versus Hinting

In this section we consider our testset of four application traces: Cscope1, Cscope2, Glimpse and Xds. We run all pairs of applications and allow each application to give hints about all accesses. The algorithms must decide when to cache versus prefetch, and how to allocate buffers between the two processes. The results are shown in Figure 7.

We consider first the three graphs in which Cscope1 runs alongside another application. Table 1 gives some statistics about the traces; it is clear that while all four have roughly similar compute times, Cscope1 places a lower I/O load on the system and thus completes much faster in isolation. The first three graphs have the same property: both TIP and LRU-SP allow the Cscope1 trace to complete substantially before the other trace, and then devote all resources to the second application.

We found that the LRU-SP policy performs poorly when it waits for a disk to go idle before issuing a batch. Instead we adopted the policy that each process waits until its current batch on a particular disk drains before sending more batches to that disk.

For the single-disk case, on average TIP2 allows the Cscope1 trace to complete

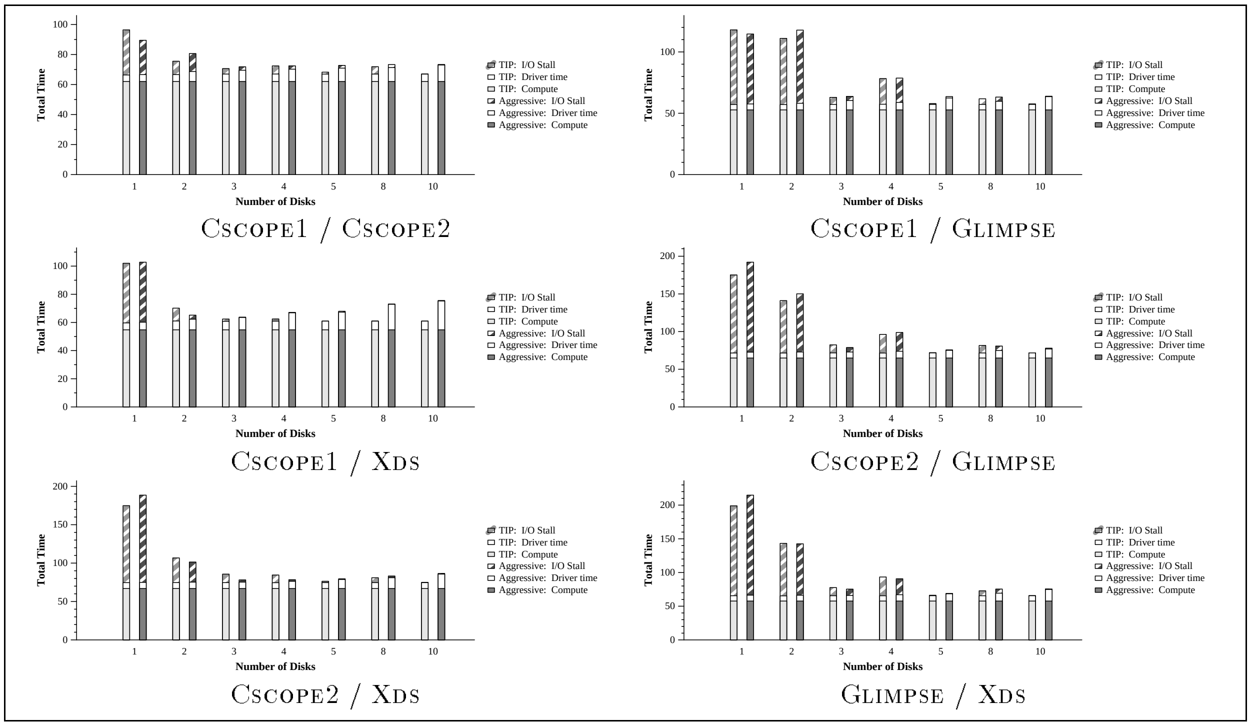


Figure 7: Two Hinting Processes

in 93% the time LRU-SP requires. TIP2 performs more caching, but LRU-SP reads deeper. When Cscope1 runs against Cscope2 with a single disk Aggressive performs 5% more fetches, but on average reads twice as deep as TIP2 into the hint streams, performing better overall.

With Cscope1 running against Xds we see the impact of driver overhead on LRU-SP's performance. The number of reads performed by LRU-SP jumps from around 10,000 with one disk to over 40,000 with ten disks; this effect occurs in the single-process case as well [KTP+96] as Aggressive will de-emphasize caching when sufficient bandwidth exists to fill the cache with future reads.

In general we see that LRU-SP tends to perform relatively better on power-of-2 numbers of disks than does TIP2. This is because load balancing is poor for some of our applications on these array sizes (notably Xds), and LRU-SP is willing to begin to prefetch a set of missing blocks on a particular disk, even if the first request to that set is far ahead.

In general the numbers indicate that performance on our small set of traces is fairly evenly split between the two systems, with perhaps a small advantage to TIP on average.

It is natural to ask whether the differences in performance are due to the cost-benefit approach versus the LRU-SP approach, or are due to fixed horizon prefetching within a process versus aggressive prefetching within a process.

Unfortunately, it is not clear how to implement aggressive prefetching in a cost-benefit framework — we are currently studying this problem. On the other hand, it is clear how to implement fixed-horizon prefetching in LRU-SP. Figure 8 shows the results of doing so, plotting the original TIP numbers against LRU-SP with fixed-horizon prefetching. Thus, the left-hand bars represent cost-benefit with fixed-horizon prefetching, and the right-hand bars are LRU-SP with fixed-horizon prefetching. These results are worse than either system described above. As shown in Figure 1, aggressive prefetching performs well on the traces we study here. On the other hand, the cost-benefit model provides good partitioning of the cache. This is one reason behind our current research efforts incorporating deeper prefetching into the cost-benefit model.

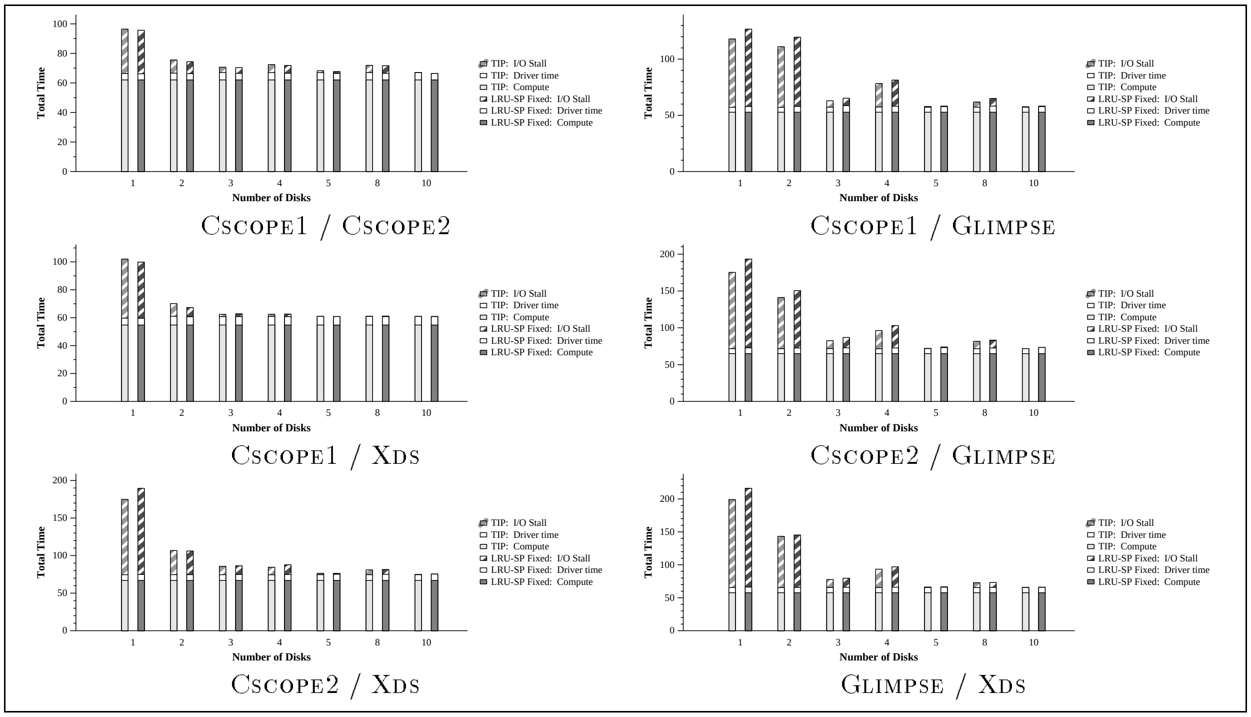
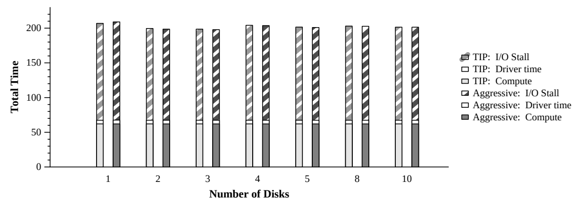


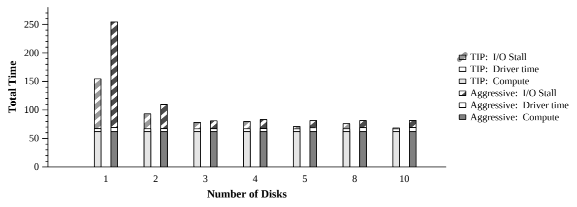
Figure 8: LRU-SP with Fixed Horizon Prefetching

5 Application Traces: Hinting versus Non-Hinting

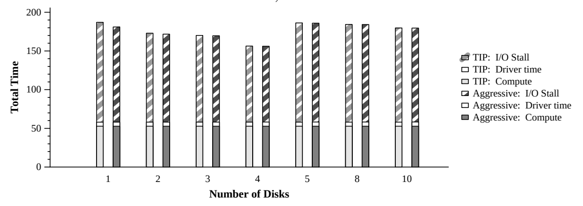
We begin by giving two graphs for each pair of distinct traces, one in which only the first process gives hints, and a second in which only the second process gives hints. The results are shown in Figures 9 and 10.



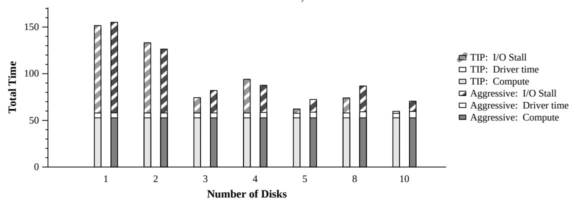
CSCOPE1 hints, CSCOPE2 nohints.



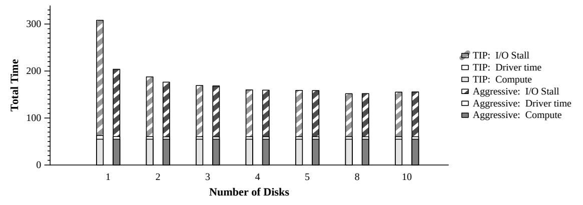
CSCOPE1 nohints, CSCOPE2 hints.



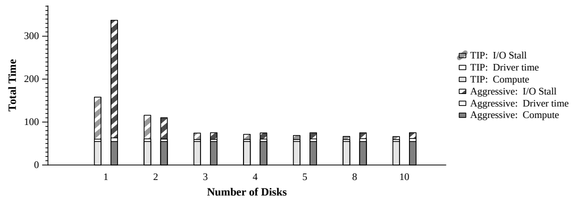
CSCOPE1 hints, GLIMPSE nohints.



CSCOPE1 nohints, GLIMPSE hints.



CSCOPE1 hints, XDS nohints.

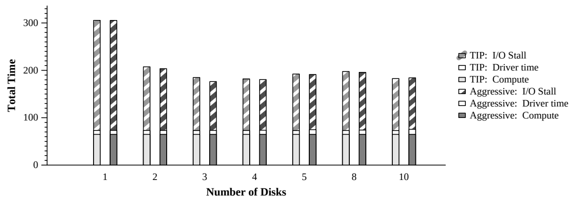


CSCOPE1 nohints, XDS hints.

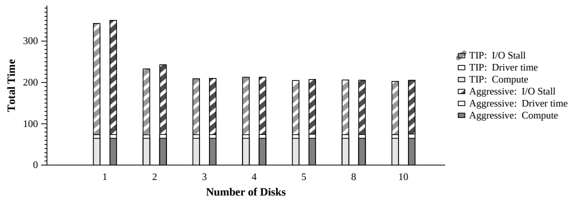
Figure 9: One Hinting Process, part 1.

Again, we begin by considering Figure 9, which shows experiments that included the CSCOPE1 trace. Consider first the graph of non-hinting CSCOPE1 versus hinting CSCOPE2. With one disk, TIP2 performs substantially better. The reason is that the CSCOPE1 working set fits in cache, and TIP2 notices and exploits this fact. On one disk, LRU-SP incurs 6600 demand misses on the non-hinting dataset, versus 2400 for TIP2.

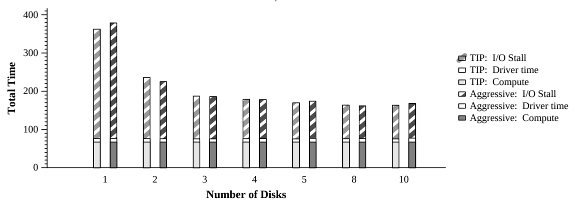
Consider alternately the graph of CSCOPE1 hinting versus XDS non-hinting. In this case, TIP2 sees an advantage in growing the LRU cache, and its assumption of sufficient bandwidth causes it to undervalue blocks that are distant in the hinted cache. Both algorithms incur similar numbers of demand misses, but LRU-SP dedicates more of the cache to the hinted sequence, performing 2100 prefetches versus 6700 for TIP2. Note that TIP2's assumption of sufficient



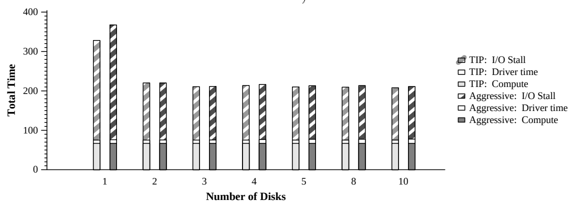
CSCOPE2 hints, GLIMPSE nohints.



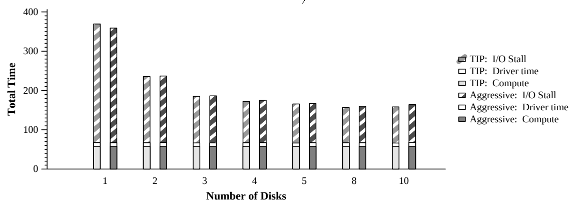
CSCOPE2 nohints, GLIMPSE hints.



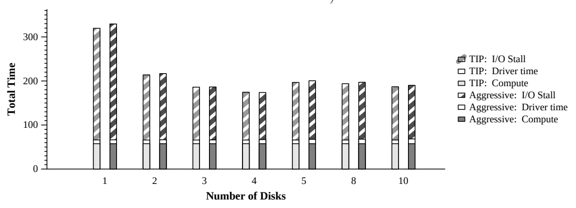
CSCOPE2 hints, XDS nohints.



CSCOPE2 nohints, XDS hints.



GLIMPSE hints, XDS nohints.



GLIMPSE nohints, XDS hints.

Figure 10: One Hinting Process, part 2.

bandwidth is the culprit in this case; if the model were extended to value distant blocks on overloaded disks more highly, this phenomenon would disappear (we are currently considering extensions in this direction). In direct opposition, when CSCOPE1 does not give hints and XDS does, LRU-SP undervalues the LRU cache and TIP2 discovers that the working set for CSCOPE1 fits in memory.

Finally, we point out that the increased timings with larger arrays in the CSCOPE1 hinting versus GLIMPSE non-hinting is due to the fact that small sequential runs lose locality as they are striped over larger arrays; by the time we reach 5 disks, the average I/O time jumps by about 20%. We expect that with a larger stripe unit than 1 block, this effect would disappear.

The remaining graphs show minor variations, but the basic lesson is that significant differences occur around discontinuities such as a working set for one process that barely fits in the cache. The graphs of Figure 10 show some variations, but all the applications are either low re-use or do not have a working set that fits in the cache.

In conclusion, when one process gives hints and the other does not we see certain situations in which each algorithm wins, though in most situations performance is similar. TIP2's cost-benefit approach may be sub-optimal if the assumption of sufficient bandwidth breaks; likewise, LRU-SP's rate-based approach may be sub-optimal if the optimal split of the cache is not proportional to the relative rates of the processes.

6 Sensitivity Analysis

In this section we perform some simple sensitivity tests on the results given above. We re-run some of the earlier experiments with different cache sizes, different processor speeds, and fifo disk queueing instead of SCAN queueing.

We begin by considering sensitivity to cache size. We choose a representative trace from above: Cscope2 versus Glimpse, with both processes hinting, and re-run the trace with a 5K buffer cache and a 15K buffer cache (initial value was 1280 buffers $\approx$ 10K). The results are shown in Figure 11. The curves are very similar to one another, and to the original graph. As shown in Section 3 there can be discontinuities in performance as cache size increases to the point that a working set will fit in core, but for typical applications prefetching ahead for relatively low re-use processes, there does not seem to be a qualitative impact.

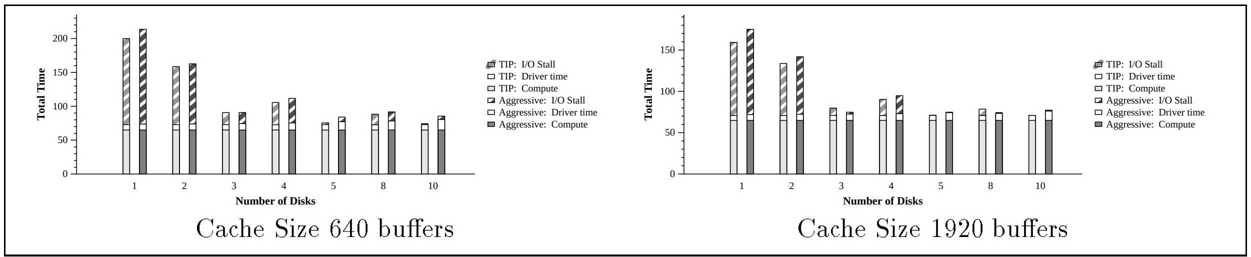
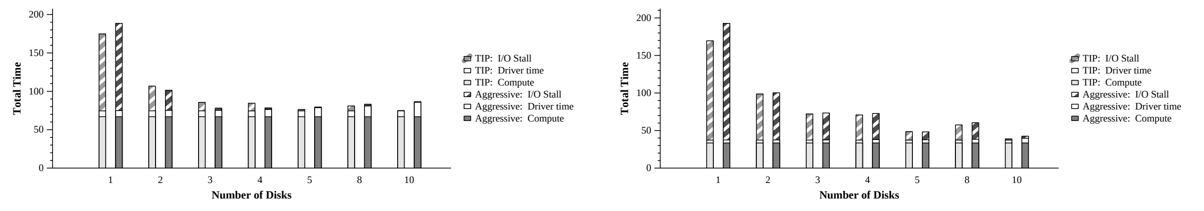


Figure 11: Sensitivity to cache size, Cscope2 versus Glimpse, both hinting.

Next, we consider today's trends towards processors that are increasingly faster than disks. We consider the Cscope2 versus Xds trace of Figure 7, in which both processes give hints. Figure 12 gives the results for a processor that is twice as fast. The only significant trend is that LRU-SP's susceptibility to driver overhead is reduced when there is still some stall in the execution (with enough bandwidth to eliminate stall, driver overhead takes the same fraction of total execution time).

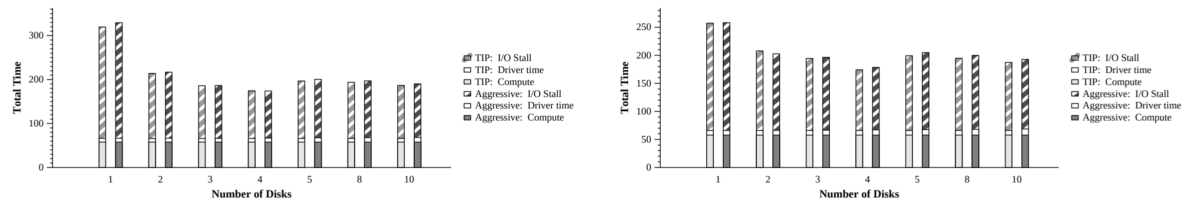
Finally, we consider the disk-head scheduling discipline. We re-run the Glimpse versus Xds trace of Figure 10, in which only Xds gives hints. The results are shown in Figure 13. Here again the results are quite similar. Across our various traces, changing the queueing discipline has unforeseen, but minor, effects on the overall timings. These results indicate that the impact of queueing is not



normal processor speed double processor speed

Figure 12: Sensitivity to processor speed, Cscope2 versus Xds, both hinting.

enormous, but may be significant; and it would be worth generating a better understanding of the relationship between prefetching and queueing.



SCAN Queueing FIFO Queueing

Figure 13: Sensitivity to queueing discipline, non-hinting Glimpse versus hinting Xds.

7 Conclusion

In this paper, we used trace-driven simulation to compare the performance of two systems that use application-level knowledge of future I/O requests to prefetch data from disk and improve the file buffer cache hit ratio. One is the informed prefetching and caching system, TIP2, developed by Patterson, Gibson, et al [PGG+95]. The other is the controlled-aggressive/LRU-SP system of Cao, Karlin, et al [CFKL95b, CFKL95a, CFL94a, CFL94b, Cao96].

There are two key issues in designing such a system. First is how to balance buffer usage between prefetching and caching for a single process' access stream. Second is how to allocate buffers among competing processes, both hinting and non-hinting. The study of [KTP+96] focuses on the first question; our study considers the second question.

In Section 3 we showed using synthetic traces that in some circumstances the cost-benefit model has significant advantages over the LRU-SP model. The

LRU-SP model allocates cache resources to processes proportionally to the rates of the processes. Conversely, the cost-benefit model estimates a locally optimal partition of the cache between processes, taking into account process rates, caching needs and hinted re-use.

In Section 4 we consider pairs of application traces running with perfect knowledge. The results indicate that TIP2's performance may be slightly superior to LRU-SP's performance, but the differences are not substantial.

Likewise, in Section 5 we consider an application with perfect knowledge running against an application with no hints. Overall we see advantages to the cost-benefit approach, but note that for our traces, the TIP2 estimator of the value of a buffer in the hinted cache may be inaccurate.

However, both algorithms actually comprise two separate components: a single-process prefetching and caching algorithm and a multi-process buffer allocation algorithm. By "TIP2", we mean the fixed-horizon single-process algorithm collaborating with the cost-benefit allocation algorithm. Likewise, by "LRU-SP" we mean the Aggressive prefetching algorithm collaborating with the LRU-SP buffer allocation algorithm. We found that LRU-SP with the fixed-horizon single-process prefetching algorithm performed worse than either of the original systems. This result, and the results of Section 3, lead us to conjecture that the cost-benefit approach has strong advantages in the multi-process problem. We are currently extending the cost-benefit approach to allow deeper, more aggressive prefetching in situations that warrant it.

We found it difficult to keep LRU-SP from thrashing the cache. In our initial implementations we found a high number of blocks being prefetched and ejected before being read. Subsequent conversations with the authors of [KTP$^+$96], and some heuristics we implemented ourselves, allowed us to cut this number down substantially. Still, our simulator and a similar simulator at the University of Washington report some thrashing at intermediate numbers of disks in the single-process case, and as we would expect, our simulator reports some non-trivial thrashing in the multiple-process case as well. However, this effect is not a significant contributor to the results described above.

This study leaves a number of issues unresolved. We adopted SCAN disk-head scheduling as a standard discipline, but we have not make a concerted effort to study the impact of the disk scheduler in a prefetching environment. [KTP$^+$96] showed the importance of good batching mechanisms for preserving locality of disk reads. We have not studied this issue carefully in the multiple-process case; we instead implemented a fixed batch-size of 16 for Aggressive. In our studies of augmenting cost-benefit analysis with deeper prefetching, we expect to incorporate a batching deep prefetcher. Finally, we have not studied the effects of incomplete or inaccurate hints. Both systems have simple mechanisms for handling such hints, but we have not examined their effectiveness.

References

[Cao96] Pei Cao. *Application-Controlled File Caching and Prefetching.* PhD thesis, Princeton University, 1996.

[CFKL95a] P. Cao, E.W. Felten, A. Karlin, and K. Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. Technical Report TR-CS95-493, Princeton University, 1995.

[CFKL95b] P. Cao, E.W. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS*, May, 1995.

[CFL94a] P. Cao, E.W. Felten, and K. Li. Application-controlled file caching policies. In *1994 Usenix Summer Technical Conference*, pages 171–182, June, 1994.

[CFL94b] P. Cao, E.W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 165–178, November, 1994.

[CP90] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 322–331. IEEE Computer Society Press, May 1990.

[KK96] T. Kimbrel and A. Karlin. Integrated parallel prefetching and caching. Technical Report UW-CSE-96-01-10, University of Washington, 1996.

[KTP$^+$96] T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E.W. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[LK91] Edward K. Lee and Randy H. Katz. Performance consequences of parity placement in disk arrays. In *ASPLOS4*, pages 190–199. ACM, 1991.

[M. 96] M. J. Acetta et al. Mach: A new kernel foundation for unix development. In *Proc. of the Summer 1996 USENIX Conference*, pages 96–113, 1996.

[MS96] L. Mummert and M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience. *Software: Practice and Experience*, 26(5), May 1996. Also available as technical report CMU-CS-94-213, School of Computer Science, Carnegie Mellon University, November 1994.

[PGG$^+$95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In SOSP 95, December, 1995.